https://doi.org/10.25559/SITITO.021.202501.25-35
УДК 004.056(045)

Оригинальная статья

Автоматизированные инструменты безопасной разработки смарт-контрактов Ethereum

А. В. Чахеев[1], **З. Р. Назаров**[2,1*]
[1] Публичное акционерное общество «Сбербанк России», г. Москва, Российская Федерация
Адрес: 117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19
[2] ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Российская Федерация
Адрес: 119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1
[*] zaxar12102001zaxar@gmail.com

Аннотация
Данная работа посвящена обзору автоматизированных инструментов безопасной разработки смарт-контрактов Ethereum. Рассматриваются актуальные уязвимости, характерные для смарт-контрактов, такие как уязвимость повторного входа, недостаточный контроль доступа, манипуляции с оракулом цены и другие. К каждой уязвимости приведена иллюстрация с уязвимым кодом. Далее рассмотрены разные типы существующих автоматизированных инструментов безопасной разработки смарт-контрактов: статический анализатор, линтер, символьный исполнитель, фаззинг и подходы на основе машинного обучения. Для каждого типа инструмента рассмотрено соответствующее реальное решение, которое является одним из лучших в своей категории. Это такие open-source решения как статический анализатор Slither, линтер Solhint, символьный исполнитель Mythril и фреймворк Foundry, который содержит в себе возможность фаззинга. Также рассмотрена текущая эффективность современных решений, которая показывает, что текущие угрозы плохо детектируется существующими инструментами. Исходя из этого предложены направления для дальнейшего развития новых инструментов безопасной разработки смарт-контрактов. Полученные результаты могут быть использованы для более глубокого понимания вопросов безопасности смарт-контрактов, а также для повышения безопасности децентрализованных приложений и развития методов автоматизированного аудита смарт-контрактов.

Ключевые слова: блокчейн, Ethereum, смарт-контракт, безопасность, уязвимости, автоматизированные инструменты

Конфликт интересов: авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Чахеев А. В., Назаров З. Р. Автоматизированные инструменты безопасной разработки смарт-контрактов Ethereum // Современные информационные технологии и ИТ-образование. 2025. Т. 21, № 1. С. 25-35. https://doi.org/10.25559/SITITO.021.202501.25-35

Original article

Automated Tools for Secure Ethereum Smart Contract Development

A. V. Chaheev[a], **Z. R. Nazarov**[b,a*]
[a] Sberbank of Russia, Moscow, Russian Federation
Address: 19 Vavilova St., Moscow 117312, Russian Federation
[b] Lomonosov Moscow State University, Moscow, Russian Federation
Address: 1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation
[*] zaxar12102001zaxar@gmail.com

Abstract
This paper provides an overview of automated tools for secure development of Ethereum smart contracts. The article discusses current vulnerabilities specific to smart contracts, such as re-entrancy vulnerability, insufficient access control, price oracle manipulation, and others. Each vulnerability is accompanied by an illustration of the vulnerable code. Next, we discuss different types of existing automated tools for secure smart contract development: static analyzer, linter, symbolic executor, fuzzing, and machine learning-based approaches. For each type of tool, a corresponding real solution is considered, which is one of the best in its category. These are open-source solutions such as the Slither static analyzer, the Solhint linter, the Mythril symbolic executor, and the Foundry framework, which includes fuzzing capabilities. The current effectiveness of modern solutions is also considered, which shows that current threats are poorly detected by existing tools. Based on this, directions for the further development of new tools for the secure development of smart contracts are proposed. The obtained results can be used to gain a deeper understanding of smart contract security issues, as well as to enhance the security of decentralized applications and develop automated smart contract auditing methods.

Keywords: blockchain, Ethereum, smart-contract, security, vulnerabilities, automated tools

Conflict of interests: The authors declare no conflict of interest.

For citation: Chaheev A.V., Nazarov Z.R. Automated Tools for Secure Ethereum Smart Contract Development. *Modern Information Technologies and IT-Education*. 2025;21(1):25-35. https://doi.org/10.25559/SITITO.021.202501.25-35

Введение

Сегодня децентрализованные приложения, построенные на смарт-контрактах, приобретают все большую популярность. Эти приложения находят широкое применение в различных областях, включая финансовые операции, управление цепочками поставок, голосование и многое другое. Смарт-контракты на платформе Ethereum позволяют автоматизировать процессы и исключить посредников, обеспечивая прозрачность и надежность выполнения соглашений.

Однако с увеличением использования смарт-контрактов растут и требования к их безопасности. Ошибки в коде смарт-контрактов могут приводить к серьезным финансовым потерям и нанесению репутационного ущерба [1, 2]. Хакеры активно ищут и эксплуатируют уязвимости, чтобы получить незаконный доступ к средствам или данным. За последние годы было множество случаев, когда атаки на уязвимости в смарт-контрактах приводили к значительным потерям, что подчеркивает необходимость разработки безопасных контрактов [3, 4]. В данной обзорной работе рассматриваются основные виды ошибок и уязвимостей, присущих смарт-контрактам на платформе Ethereum, а также методы и средства, которые помогают разработчикам создавать более безопасные смарт-контракты.

1. Основные понятия

Блокчейн

Блокчейн представляет собой последовательность блоков, каждый из которых содержит информацию о транзакциях. Эти блоки связаны друг с другом с использованием криптографии, что обеспечивает неизменность данных: каждый новый блок включает криптографический хэш предыдущего блока, создавая непрерывную и защищённую цепочку. Описанная структура становится инновационной лишь при условии дублирования цепочки на множестве вычислительных узлов, каждый из которых хранит одну копию [5]. При наличии дубликатов цепочка становится децентрализованной, эту абстракцию часто называют децентрализованной бухгалтерской книгой. Именно о такой децентрализованной цепочке идёт речь, когда используется термин «блокчейн».

Для включения новой транзакции клиент создает ее и отправляет на один или несколько узлов блокчейн-сети, затем транзакция распространяется по сети. Узлы собирают актуальные транзакции, группируют их в блок и предлагают этот блок сети в соответствии с установленными правилами. Далее механизм консенсуса обеспечивает согласование всех узлов системы относительно общего состояния блокчейна. После достижения соглашения по блоку соответствующие узлы добавляют его в свою цепочку блоков.

Ethereum

Ethereum является одной из ведущих платформ для разработки и развертывания децентрализованных приложений[1]. Она была предложена Виталиком Бутериным в конце 2013 года и запущена в 2015 году. Ethereum предоставляет мощную и гибкую инфраструктуру для создания смарт-контрактов и децентрализованных приложений, обеспечивая децентрализованное выполнение программного кода на основе технологии блокчейн. Ethereum использует механизм консенсуса proof of stake.

Основным компонентом сети Ethereum является виртуальная машина Ethereum (EVM, Ethreum Virtual Machine). Она выполняет байт-код смарт-контрактов, который генерируется при компиляции исходного кода на высокоуровневом языке Solidity. Кроме того, EVM поддерживает концепцию «газа», которая представляет собой стоимость вычислительных ресурсов, необходимых для выполнения операций. Это стимулирует эффективное использование ресурсов и предотвращает бесконечные циклы, способные вывести из строя сеть [6, 7]. EVM гарантирует, что все узлы сети могут достигать консенсуса по состоянию контрактов и результатам их выполнения, обеспечивая согласованность и целостность блокчейна Ethereum.

Смарт-контракт

Смарт-контракт – это совокупность кода (его функций) и данных (его состояния), которые находятся по определенному адресу в блокчейне Ethereum. Смарт-контракт регулирует выполнение заранее прописанного соглашения между двумя или более сторонами в цифровой среде. Отличие таких цифровых соглашений от традиционных в том, что для них не требуется третья доверенная сторона.

Смарт-контракт Ethereum, как правило, пишется на высокоуровневом языке программирования Solidity, затем он проходит этап компиляции, превращаясь в байт-код. Компилятор Solidity преобразует исходный код в EVM байт-код и генерирует ABI (Application Binary Interface)[2], описывающий интерфейс контракта для взаимодействия с внешними приложениями.

Скомпилированный байт-код включается в специальную транзакцию без адресата, которая отправляется в сеть Ethereum и ждет подтверждения. Когда транзакция включается в новый блок блокчейна, виртуальная машина Ethereum (EVM) выполняет байт-код, и смарт-контракт развертывается на блокчейне, получая уникальный адрес и становясь доступным для взаимодействия с другими контрактами и участниками сети.

Взаимодействие со смарт-контрактом на платформе Ethereum осуществляется через транзакции. Пользователь отправляет транзакцию, в которой указывает адрес контракта, селектор и параметры вызываемой функции, а также количество эфира, если это требуется. Когда транзакция включается в блок и обрабатывается виртуальной машиной Ethereum (EVM), функция контракта выполняется с переданными параметрами. В результате выполнения функции происходит изменение состояния контракта или возвращается необходимая информация, которая может быть прочитана из блокчейна [8].

[1] Buterin V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform [Электронный ресурс] // Ethereum Whitepaper, 2025. URL: https://ethereum.org/en/whitepaper/ (дата обращения: 25.02.2025).

[2] Contract ABI Specification – Solidity 0.8.30 documentation. 2025 [Электронный ресурс]. URL: https://docs.soliditylang.org/en/latest/abi-spec.html (дата обращения: 25.02.2025).

2. Основные угрозы и уязвимости смарт-контрактов

Рассмотрим некоторые самые распространенные ошибки по версии OWASP за 2025 год[3].

2.1 Access Control Vulnerabilities

Публичную («public») или внешнюю («external») функцию смарт-контракта в Ethereum можно вызвать с любого адреса. Обычно доступ к функции в смарт-контракте ограничивается через проверку «msg.sender» (адрес вызывающего смарт-контракт) на допустимый адрес. При отсутствии должной проверки полномочий пользователя злоумышленники могут получить несанкционированный доступ к данным или функциям контракта, что особенно опасно при наличии критически важных операций, таких как эмиссия или сжигание токенов, голосование, вывод средств, а также операции по обновлению или изменению владения контрактом [9].

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_AccessControl {
    mapping(address => uint256) public balances;

    // Burn function with no access control
    function burn(address account, uint256 amount) public {
        _burn(account, amount);
    }
}
```

Р и с. 1. Код с уязвимой функцией burn[4]
F i g. 1. Code with vulnerable burn function[4]

В данном примере, представленного на рисунке 1, функцию по сжиганию токенов может вызвать любой, что может привести к финансовым потерям.
Данная уязвимость является самой эксплуатируемой за 2024 год: 953.2 миллиона долларов США было утеряно.

2.2 Price Oracle Manipulation

Оракулы – это внешние источники данных, которые передают смарт-контрактам актуальную информацию, такую как цены активов или другие рыночные показатели. В экосистеме децентрализованных финансов они играют ключевую роль, обеспечивая корректную работу смарт-контрактов. Следовательно, уязвимость к манипуляции оракулом цен представляет собой серьезную проблему: если злоумышленник подделает или исказит передаваемые данные, это может привести к некорректному функционированию контрактов и значительным финансовым потерям.

Основная проблема заключается в том, что злоумышленники могут манипулировать данными, получаемыми от оракула. При отсутствии достаточных механизмов проверки подлинности и корректности этих данных контракт может работать на основе искаженной информации. Это позволяет искусственно завышать или занижать стоимость активов, получать займы, превышающие реальную стоимость залога, приводить к принудительной ликвидации законных пользователей из-за ложной оценки их залоговых активов, а также опустошать ликвидные пулы, манипулируя данными для получения несанкционированного доступа к средствам.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IPriceFeed {
    function getLatestPrice() external view returns (int);
}

contract PriceOracleManipulation {
    address public owner;
    IPriceFeed public priceFeed;

    constructor(address _priceFeed) {
        owner = msg.sender;
        priceFeed = IPriceFeed(_priceFeed);
    }

    function borrow(uint256 amount) public {
        int price = priceFeed.getLatestPrice();
        require(price > 0, "Price must be positive");

        // Vulnerability: No validation or protection against price manipulation
        uint256 collateralValue = uint256(price) * amount;

        // Borrow logic based on manipulated price
        // If an attacker manipulates the oracle, they could borrow more than they should
    }

    function repay(uint256 amount) public {
        // Repayment logic
    }
}
```

Р и с. 2. Манипуляция ценовым оракулом[5]
F i g. 2. Price Oracle Manipulation[5]

На рисунке 2 приведен пример кода уязвимого к манипуляции ценой через оракула. Если оракул предоставит нам неверное значение цены, то пользователь, вызвав функцию «borrow», сможет взять кредит под залог любой суммы (при соответствующем неверном значении цены).
Данная уязвимость появилась в TOP-10 OWASP 2025 и заняла второе место по распространенности: 8.8 миллиона долларов США было утеряно в результате манипуляций оракулами.

2.3 Logic Errors

Ошибки бизнес-логики представляют собой труднонаходимые недочеты в реализации смарт-контрактов, когда фактическое поведение кода не соответствует изначально задуманной логике. Такие ошибки могут проявляться в различных аспектах работы контракта, включая некорректное распределение наград, ошибки в механизмах выпуска токенов или неправильные расчеты при осуществлении операций заимствования и возврата средств.

[3] OWASP Smart Contract Top 10 [Электронный ресурс] // OWASP Foundation, 2025. URL: https://owasp.org/www-project-smart-contract-top-10 (дата обращения: 25.02.2025).

[4] SC01:2025 – Improper Access Control [Электронный ресурс] // OWASP Foundation, 2025. URL: https://owasp.org/www-project-smart-contract-top-10/2025/en/src/SC01-access-control.html (дата обращения: 25.02.2025).

[5] SC02:2025 – Price Oracle Manipulation [Электронный ресурс] // OWASP Foundation, 2025. URL: https://owasp.org/www-project-smart-contract-top-10/2025/en/src/SC02-price-oracle-manipulation.html (дата обращения: 25.02.2025).

```
1  contract ERC20 {
2    // owner => spender => amount
3    mapping (address => mapping (address => uint256))
4      internal _allowances;
5
6    function _approve(address owner, address spender,
7      uint256 allowance) internal {
8      _allowances[owner][spender] = allowance;
9    }
10
11   function transferFrom(address from, address to,
12     uint256 amount) external {
13     require(_allowances[from][msg.sender] >= amount);
14     _approve(from, msg.sender,
15       _allowances[from][to] - amount);
16     _transfer(from, to, amount);
17   }
18 }
```

Р и с. 3. Использование Redacted Cartel [10]

F i g. 3. The Redacted Cartel exploit [10]

Контракт токена на рисунке 3 содержит ошибку бизнес-логики, которая позволяет злоумышленнику незаконно увеличить свой лимит разрешённых токенов. В коде есть отображение «_allowances», хранящее разрешённые суммы токенов. Функция «_approve()» обновляет их, а «transferFrom» переводит токены между адресами. Ошибка возникла из-за обновления разрешения для адреса «to» вместо «msg.sender» (строка 15). Это позволяет злоумышленнику вызвать «transferFrom(Alice, Bob, 0)», из-за чего её лимит неправомерно увеличивался, хотя токены не переводились. Аудитор, обнаруживший уязвимость, получил награду в 560 000 долларов США.

Ошибки бизнес-логики является одними из самых трудно находимых ошибок, как автоматизированными средствами, так и с помощью ручного аудита [11].

2.4 Reentrancy Attacks

Атака повторного входа представляет собой одну из наиболее известных уязвимостей в смарт-контрактах. Проблема возникает, когда функция смарт-контракта выполняет внешний вызов к другому контракту до того, как завершится обновление её внутреннего состояния. Это позволяет злоумышленнику, посредством специально сконструированного внешнего контракта, повторно войти в исходную функцию и выполнить её повторно с использованием прежнего, ещё не изменённого состояния. В результате такой атаки возможно несанкционированное многократное снятие средств, что может привести к полному опустошению баланса контракта.

На рисунке 4 представлен код уязвимого контракта Victim и код атакующего Attacker. Алгоритм атаки следующий: атакующий вызывает функцию «startAttack» с адресом жертвы, эта функция вызывает первый раз функцию «withdraw» у Victim, затем Victim переводит токены на Attacker, это вызывает fallback-функцию (ее определение находится в 31 строчке), которая вызывает второй раз «withdraw» и так далее пока count не достигнет значения 10. Fallback-функция вызывается каждый раз до того, как обнуляется счет атакующего (20-ая строчка).

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  contract Victim {
5      mapping(address => uint) public balances;
6
7      function deposit() external payable {
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdraw() external {
12         uint amount = balances[msg.sender];
13         require(amount > 0, "Insufficient balance");
14
15         // Vulnerability: Ether is sent before updating the user's balance, allowing reentrancy.
16         (bool success, ) = msg.sender.call{value: amount}("");
17         require(success, "Transfer failed");
18
19         // Update balance after sending Ether
20         balances[msg.sender] = 0;
21     }
22 }
23
24 contract Attacker{
25     uint count = 0;
26
27     function startAttack(victimAddress){
28         Victim(victimAddress).withdraw();
29     }
30
31     function() payable{
32         if (++count < 10){
33             Victim(msg.sender).withdraw();
34         }
35     }
36 }
```

Р и с. 4. Атака повторного входа[6]

F i g. 4. Reentrancy attack[6]

Эта уязвимость изучена исследователями достаточно подробно [12-15], а также большинство автоматизированных инструментов безопасной разработки смарт-контрактов умеет детектировать атаку повторного входа.

Основной принцип, который помогает защититься от атаки повторного входа это соблюдение «checks-effects-interactions» при ситуациях обработки переменных состояния.

2.5 Lack of Input Validation

Валидация входных данных представляет собой критически важный этап в работе смарт-контракта, который гарантирует, что обрабатываемые данные соответствуют ожидаемым параметрам. Недостаточная проверка входных данных может привести к тому, что контракт будет работать с некорректными или вредоносными значениями, что создаёт благоприятные условия для атак и нарушений логики работы. Эта уязвимость может использоваться для внесения вредоносных данных, манипуляции логикой работы контракта и неавторизованного изменения переменных состояния.

На рисунке 5 представлен код, который описывает два взаимодействующих контракта: «Treasure» и «BadInvestor». Допустим на контракте «BadInvestor» находится какое-то количество ETH и владелец контракта хочет инвестировать их «Treasure». Для этого он вызывает функцию «invest» с настоящим адресом «Treasure» и суммой инвестиции. Злоумышленник может также вызвать «invest» только с уже своим адресом, который реализует интерфейс «Treasure», что переводит деньги на контракт злоумышленника. Злоумышленник может так сделать, так как внутри смарт-контракта нет никакой проверки на корректность адреса «Treasure».

Таким образом, отсутствие должной проверки может приве-

[6] SC05:2025 – Reentrancy [Электронный ресурс] // OWASP Foundation, 2025. URL: https://owasp.org/www-project-smart-contract-top-10/2025/en/src/SC05-reentrancy-attacks.html (дата обращения: 25.02.2025).

сти к финансовым потерям, нарушению целостности данных и общему снижению надежности системы.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Treasure{
    uint treasure;
    mapping(address => uint256) public investors;

    function funding() payable{
        treasure += msg.value;
        investors[msg.sender] = msg.value;
    }
}

contract BadInvestor {
    mapping(address => uint256) public balances;

    // Invest function with no input validation
    function invest(address treasure, uint256 amount) {
        require(address(this).balance >= amount, "Insufficient balance");

        //Calling an external smart contract at an address that has not been validated
        Treasure(treasure).funding{value: amount}();
    }

    receive() external payable {}
}
```

Р и с. 5. Пример отсутствия валидации входных параметров функции

F i g. 5. An example of lack of validation of function input parameters

Источник: составлено авторами.

Source: Compiled by the authors.

3. Инструменты безопасной разработки смарт-контрактов

Разработка смарт-контрактов требует особого внимания к безопасности, поскольку ошибки в коде могут привести к значительным финансовым потерям [16]. Уязвимости необходимо своевременно выявлять и устранять, используя как ручные, так и автоматизированные методы анализа. Ручной аудит позволяет экспертам глубже погружаться в логику кода, но является трудозатратным и подвержен человеческому фактору. В то же время автоматизированные инструменты значительно ускоряют процесс обнаружения уязвимостей, повышая его эффективность. В данной главе рассмотрены основные виды автоматических средств, предназначенных для анализа и повышения безопасности смарт-контрактов Ethereum.

3.1 Статический анализатор

Статический анализатор для Solidity — это инструмент, предназначенный для автоматического анализа смарт-контрактов без их выполнения с целью выявления уязвимостей, оптимизации и нарушений лучших практик программирования. Он работает на основе анализа исходного кода, используя статические методы проверки.

Использование статических анализаторов помогает разработчикам писать более безопасные и эффективные смарт-контракты.

На сегодняшний день они являются лидерами по нахождению уязвимостей среди автоматических инструментов безопасной разработки смарт-контрактов.

В сравнении с другими инструментами безопасной разработки, статические анализаторы обычно имеют преимущество в скорости работы, хорошо находят простые и хорошо изученные уязвимости. Из недостатков можно отметить большое количество ложных срабатываний.

Slither. Один из самых популярных статических анализаторов Solidity это open-source проект Slither[7], который написан на языке Python. Он содержит в себе около 100 детекторов, таких как suicidal (обнаруживает конструкции, позволяющие уничтожить контракт), reentrancy-eth (обнаруживает уязвимости повторного входа), controlled-delegatecall (анализатор уязвимостей связанных с delegatecall), tx-origin (определение опасного использования tx.origin) и другие.

Также Slither легко интегрируется с популярными фреймворками для разработки смарт-контрактов, такими как Foundry и Hardhat. Среднее время выполнения тестов — менее одной секунды на контракт (в зависимости от сложности контракта). В работе [11] отмечается, что Slither обнаружил уязвимости в реальных смарт-контрактах на сумму 149 792 690 долларов США из 271 553 041 долларов США. При этом другие инструменты не обнаружили уязвимостей, или обнаружили на сумму в 6 раз меньше, чем Slither.

3.2 Линтер

Линтер – это программный инструмент для интегрированной среды разработки, предназначенный для статического анализа кода налету. Основные функции линтера это выявление синтаксических ошибок в коде смарт-контрактов и ошибок, которые могут привести к уязвимостям. Также инструмент проверяет код на соответствие установленным рекомендациям по стилю программирования, что способствует поддержанию единообразия и читаемости кода. Также у него присутствует возможность гибкой настройки набора проверяемых правил в зависимости от потребностей проекта.

Solhint. Одним из наиболее распространённых линтеров для Solidity является open-source проект Solhint[8]. Он обладает всеми описанными выше свойствами. Вот некоторые правила безопасности, которые он использует: reentrancy (правило, направленное на поиск уязвимости повторного входа), func-visibility (правило, которое выдает программисту рекомендацию на явное указание видимости функции), multiple-sends (правило, детектирующее многократных вызовы send в одной транзакции, что небезопасно) и другие. Также присутствует множество правил лучших практик, правил стиля программирования и правил по использованию газа.

Использование линтеров значительно снижает вероятность ошибок и уязвимостей на ранних этапах разработки. Они позволяют автоматизировать процесс анализа кода, обеспечивая его соответствие лучшим практикам программирования. В совокупности с другими инструментами безопасной разработки, такими как статический и динамический анализаторы, линтеры играют важную роль в обеспечении надежности и безопасности смарт-контрактов Ethereum.

[7] Static Analyzer for Solidity and Vyper [Электронный ресурс] // GitHub, 2025. URL: https://github.com/crytic/slither (дата обращения: 25.02.2025).

[8] Solhint is an open-source project to provide a linting utility for Solidity code [Электронный ресурс] // GitHub, 2025. URL: https://github.com/protofire/solhint (дата обращения: 25.02.2025).

3.3 Символический исполнитель

Одним из мощных методов анализа программ является символьное исполнение (symbolic execution). Этот метод широко используется для автоматизированного поиска уязвимостей в смарт-контрактах Solidity, позволяя исследовать различные пути выполнения кода и выявлять потенциальные риски до их развертывания в сети Ethereum.

Символьное исполнение позволяет анализировать поведение смарт-контракта, заменяя конкретные входные данные на символические значения. Такой подход позволяет инструменту учитывать все возможные варианты выполнения кода и выявлять потенциальные уязвимости. Таким образом, эта методика сочетает в себе элементы динамического и статического анализатора. Основные этапы работы символьного исполнителя:

1. Использование символьных переменных – вместо конкретных значений вводные параметры рассматриваются как символические переменные, что позволяет анализировать различные возможные состояния контракта.
2. Исследование путей выполнения – инструмент последовательно анализирует все возможные пути выполнения смарт-контракта, подставляя символические значения.
3. Оценка условий и ветвлений – выражения, содержащие условия, циклы и другие логические конструкции, анализируются с точки зрения возможных вариантов исполнения.
4. Выявление уязвимостей – на основании полученных данных инструмент обнаруживает потенциальные проблемы, такие как атаки повторного входа и переполнения.

Mythril. Одним из наиболее популярных инструментов символьного исполнения для анализа смарт-контрактов на Solidity является Mythril[9]. Этот инструмент анализирует байткод и выявляет уязвимости в смарт-контрактах, созданных для Ethereum и других блокчейнов, совместимых с EVM.

Внутри себя Mythril содержит такие модули, как «delegate call to untrusted contract» (выявляет опасное использование delegatecall), «dependence on predictable variables» (анализирует зависимости от манипулируемых данных), «external calls» (анализирует риски атак повторного входа), «multiple sends» (предотвращает отказ в обслуживании из-за множественных отправок), «state change external calls» (анализирует изменения состояния после внешних вызовов), «unchecked retval» (проверяет корректность обработки возвращаемых значений), «user supplied assertion» (выявляет потенциально опасные пользовательские проверки), «arbitrary storage write» (анализирует возможность записи в произвольные слоты хранилища) и «arbitrary jump» (выявляет случаи контролируемых пользователем переходов в коде).

3.4 Фаззинг

Фаззинг является одним из ключевых методов автоматизированного тестирования безопасности смарт-контрактов Ethereum. Этот подход основан на генерации случайных, неожиданных или преднамеренно некорректных входных данных, с целью выявления потенциальных уязвимостей и нежелательного поведения контрактов. В отличие от традиционных методов тестирования, он позволяет исследовать более широкий спектр возможных состояний контракта, что особенно важно в среде децентрализованных финансов, где даже небольшие ошибки могут привести к значительным финансовым потерям.

Основной целью фаззинга является проверка инвариантов – критически важных условий, которые должны оставаться неизменными при любом сценарии выполнения контракта. Фаззинг помогает выявить ошибки, связанные с некорректной валидацией входных данных, нежелательными изменениями состояния контракта, нарушением логики выполнения транзакций и другие.

Foundry Fuzz Testing. Open-source проект Foundry[10] – это набор инструментов для разработки смарт-контрактов на Ethereum. В Foundry фаззинг интегрирован в процесс тестирования. При написании тестов с использованием Forge, компонента Foundry, можно определить функции с параметрами, и Forge автоматически будет генерировать случайные значения для этих параметров при выполнении тестов. Foundry предоставляет возможность настройки параметров фаззинга через файл конфигурации или переменные окружения. Например, можно задать количество итераций фаззинга или использовать определенный seed для генерации случайных чисел, что обеспечивает воспроизводимость тестов.

3.5 Инструменты на основе машинного обучения

Инструменты на основе машинного обучения представляют собой перспективное направление в области анализа безопасности смарт-контрактов Ethereum. Они способны автоматизировать процесс выявления уязвимостей и повышать точность анализа. Однако для их эффективного использования необходимо учитывать ограничения, связанные с необходимостью качественных данных для обучения моделей и невозможностью обнаружения новых типов уязвимостей. В сочетании с традиционными методами анализа DL-инструменты могут значительно повысить уровень безопасности смарт-контрактов.

Последние инструменты на основе глубокого обучения, используют языковые модели (LLM) [17] для анализа кода смарт-контрактов: SymGPT [18] сочетает символическое исполнения с LLM для проверки соответствия смарт-контрактов стандартам ERC [19], ContractTinker [20] использует подход Chain-of-Thought (CoT) для генерации исправлений уязвимостей, LLM-SmartAudit [21] применяет многоагентный подход для анализа кода, SCALM [22] фокусируется на обнаружении плохих практик, используя методологию Retrieval-Augmented Generation (RAG).

Основные ограничения DL-инструментов связаны с высокой требовательностью к данным и ограниченной способностью выявлять новые уязвимости, отсутствующие в обучающей выборке. Эти факторы снижают их эффективность по сравнению с классическими методами статического и динамического анализа.

[9] Mythril is a symbolic-execution-based securty analysis tool for EVM bytecode [Электронный ресурс] // GitHub, 2025. URL: https://github.com/Consensys/mythril (дата обращения: 25.02.2025).

[10] Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust [Электронный ресурс] // GitHub, 2025. URL: https://github.com/foundry-rs/foundry (дата обращения: 25.02.2025).

4. Эффективность инструментов безопасной разработки смарт-контрактов

Несмотря на существование множества инструментов автоматического анализа безопасности, их эффективность в предотвращении реальных атак остается под вопросом. В данной главе описывается актуальная эффективность современных инструментов безопасной разработки смарт-контрактов и выявляются их ограничения

4.1 Охват уязвимостей

Исследования показывают, что современные инструменты автоматизированного анализа способны обнаруживать только ограниченный набор уязвимостей. Например, в исследовании [11] было выявлено, что лишь 20% эксплуатируемых ошибок можно обнаружить с помощью существующих инструментов. Среди наиболее детектируемых уязвимостей – уязвимость повторного входа (reentrancy) и ошибки, связанные с областью видимости функций.

Значительная часть ошибок остается вне зоны охвата инструментов. В исследовании [10] отмечается, что автоматические анализаторы неэффективны против логических ошибок и манипуляций с оракулами, которые составляют до 75% реальных атак.

4.2 Вклад в предотвращение атак

Анализ 127 реальных атак, проведенный в работе [10], показал, что существующие инструменты могли бы предотвратить только 8% атак, что эквивалентно 149 миллионов долларов США потерь из 2,3 миллиардов. Основной проблемой является неспособность инструментов обнаруживать уязвимости, связанные с бизнес-логикой и специфическими реализациями контрактов.

В работе [11] показано, что 80% уязвимостей, используемых в атаках, не могут быть обнаружены существующими автоматизированными инструментами.

4.3 Пути повышения эффективности

Повышение эффективности автоматизированных инструментов анализа безопасности смарт-контрактов требует развития новых подходов, поскольку существующие методы выявляют лишь ограниченный набор уязвимостей и слабо предотвращают реальные атаки.

Ключевыми направлениями совершенствования являются развитие семантического анализа бизнес-логики контрактов, использование методов машинного обучения для поиска уязвимостей, а также развитие полуавтоматических средств безопасной разработки смарт-контрактов [23-25].

Заключение

В данной работе проведен обзор актуальных проблем безопасности смарт-контрактов на платформе Ethereum, рассмотрены наиболее распространенные уязвимости, а также автоматизированные инструменты, предназначенные для их выявления. Были рассмотрены статические анализаторы, линтеры, инструменты символьного исполнения, фаззеры и решения на основе машинного обучения.

Актуальные работы по эффективности инструментов безопасной разработки показали, что, несмотря на наличие широкого спектра инструментов, их результативность в предотвращении атак ограничена. Современные методы хорошо справляются с обнаружением классических уязвимостей, таких как атаки повторного входа или некорректная валидация входных данных, но остаются уязвимыми перед сложными логическими ошибками и манипуляциями с оракулами. Это указывает на необходимость разработки новых гибридных подходов, объединяющих различные методы анализа и машинного обучения. Автоматизированные инструменты безопасности играют ключевую роль в улучшении надежности смарт-контрактов, но их использование должно сочетаться с ручными аудитами и лучшими практиками разработки. Необходимы дальнейшие исследования в этой области, для того чтобы повысить уровень безопасности децентрализованных приложений и доверие к блокчейн-технологиям в целом.

Благодарности

Настоящая работа подготовлена по результатам исследований, проведённых при выполнении магистерской диссертации на факультете вычислительной математики и кибернетики МГУ имени М. В. Ломоносова в рамках совместной образовательной программы ПАО Сбербанк – магистратуры «Кибербезопасность».

Список использованных источников

[1] Абрамов В. И., Глазков А. А. Перспективы использования смарт-контрактов в развитии бизнес-экосистем // Экономика. Информатика. 2022. Т. 49, № 2. С. 256-267. https://doi.org/10.52575/2687-0932-2022-49-2-256-267

[2] Луценко С. И. Роль смарт-контрактов в современных цифровых реалиях // Цифровая экономика. 2021. № 2(14). С. 37-41. https://doi.org/10.34706/DE-2021-02-05

[3] Allam Z. On Smart Contracts and Organisational Performance: A Review of Smart Contracts Through The Blockchain Technology // Review of Economic and Business Studies. 2018. Vol. 11, issue 2. P. 137-156. https://doi.org/10.1515/rebs-2018-0078

[4] An overview on smart contracts: Challenges, advances and platforms / Z. Zheng [et al.] // Future Generation Computer Systems. 2020. Vol. 105. P. 475-491. https://doi.org/10.1016/j.future.2019.12.019

[5] Blockchain: A Crypto-Intensive Technology – A Comprehensive Review / M. I. Sarwar [et al.] // IEEE Access. 2023. Vol. 11. P. 141926-141955. https://doi.org/10.1109/ACCESS.2023.3342079

[6] Ethereum Transaction Replay Platform Based on State-Wise Account Input Data / Y. Huang [et al.] // IEEE Transactions on Services Computing. 2024. Vol. 17, issue 5. P. 2404-2416. https://doi.org/10.1109/TSC.2024.3390433

[7] Жихарев А. Г., Киданов В. В., Корсунов Н. И. Системно-объектное моделирование смарт-контрактов // Экономика. Информатика. 2023. Т. 50, № 4. С. 859-872. https://doi.org/10.52575/2712-746X-2023-50-4-859-872

[8] Ivanov N., Yan Q., Kompalli A. TxT: Real-Time Transaction Encapsulation for Ethereum Smart Contracts // IEEE Transactions on Information Forensics and Security. 2023. Vol. 18. P. 1141-1155. https://doi.org/10.1109/TIFS.2023.3234895

[9] Shi J., Li R., Hou W. A Mechanism to Resolve the Unauthorized Access Vulnerability Caused by Permission Delegation in Blockchain-Based Access Control // IEEE Access. 2020. Vol. 8. P. 156027-156042. https://doi.org/10.1109/ACCESS.2020.3018783

[10] Demystifying Exploitable Bugs in Smart Contracts / Z. Zhang [et al.] // 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE Press, 2023. P. 615-627. https://doi.org/10.1109/ICSE48619.2023.00061

[11] Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners? / S. Chaliasos [et al.] // Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). New York, NY, USA: ACM, 2024. Article number: 60. P. 1-13. https://doi.org/10.1145/3597503.3623302

[12] Samreen N. F., Alalfi M. H. Reentrancy Vulnerability Identification in Ethereum Smart Contracts // 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). London, ON, Canada: IEEE Press, 2020. P. 22-29. doi: https://doi.org/10.1109/IWBOSE50093.2020.9050260

[13] ReGuard: finding reentrancy bugs in smart contracts / C. Liu [et al.] // Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18). New York, NY, USA: ACM, 2018. P. 65-68. https://doi.org/10.1145/3183440.3183495

[14] Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models / P. Qian [et al.] // IEEE Access. 2020. Vol. 8. P. 19685-19695. https://doi.org/10.1109/ACCESS.2020.2969429

[15] Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts / Y. Xue [et al.] // 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, VIC, Australia: IEEE Press, 2020. P. 1029-1040.

[16] Challenges and Common Solutions in Smart Contract Development / N. Kannengießer // IEEE Transactions on Software Engineering. 2022. Vol. 48, issue 11. P. 4291-4318. https://doi.org/10.1109/TSE.2021.3116808

[17] Large Language Models in Cyberattacks / S. V. Lebed, D. E. Namiot, E. V. Zubareva [et al.] // Doklady Mathematics. 2024. Vol. 110(Suppl 2). P. S510-S520. https://doi.org/10.1134/S1064562425700012

[18] SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models / S. Xia [et al.] // arXiv:2502.07644. 2025. https://doi.org/10.48550/arXiv.2502.07644

[19] Formal Verification of ERC-Based Smart Contracts: A Systematic Literature Review / R. B. Fekih [et al.] // IEEE Access. 2025. Vol. 13. P. 11396-11422. https://doi.org/10.1109/ACCESS.2025.3527158

[20] ContractTinker: LLM-Empowered Vulnerability Repair for Real-World Smart Contracts / C. Wang [et al.] // 2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE). Sacramento, CA, USA: IEEE Press, 2024. P. 2350-2353.

[21] LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection / Z. Wei [et al.] // arXiv:2410.09381. 2024. https://doi.org/10.48550/arXiv.2410.09381

[22] SCALM: Detecting Bad Practices in Smart Contracts Through LLMs / Z. Li [et al.] // arXiv:2502.04347. 2025. https://doi.org/10.48550/arXiv.2502.04347

[23] Намиот Д. Е., Сухомлин В. А. О кибербезопасности систем интернета вещей // International Journal of Open Information Technologies. 2023. Т. 11, № 2. С. 85-97. EDN: DCDCHM

[24] Намиот Д. Е., Куприяновский В. П. Архитектурные модели Web3 // International Journal of Open Information Technologies. 2024. Т. 12, № 2. С. 84-95. EDN: CDXZIS

[25] Умная инфраструктура, физические и информационные активы, Smart Cities, BIM, GIS и IoT / В. П. Куприяновский, В. В. Аленьков, И. А. Соколов [и др.] // International Journal of Open Information Technologies. 2017. Т. 5, № 10. С. 55-86. EDN: ZISODV

Поступила 25.02.2025; одобрена после рецензирования 28.03.2025; принята к публикации 16.04.2025.

Об авторах:

Чахеев Андрей Валерьевич, директор Департамента кибербезопасности, Публичное акционерное общество «Сбербанк России» (117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19), **ORCID: https://orcid.org/0009-0003-2299-5416**, AVChaheev@sberbank.ru

Назаров Захар Романович, студент совместной магистратуры «Кибербезопасность МГУ-СБЕР» факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1); аналитик Центра управления продуктами кибербезопасности Департамента кибербезопасности, Публичное акционерное общество «Сбербанк России» (117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19), **ORCID: https://orcid.org/0009-0004-5276-4253**, zaxar12102001zaxar@gmail.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

References

[1] Abramov V.I., Glazkov A.A. Prospects for the Use of Smart Contracts in the Development of Business Ecosystems. *Economics. Information technologies*. 2022;49(2):256-267. (In Russ., abstract in Eng.) https://doi.org/10.52575/2687-0932-2022-49-2-256-267

[2] Lutsenko S.I. The role of smart contracts in modern digital realities. *Digital Economy***.** 2021;(2):37-41. (In Russ., abstract in Eng.) https://doi.org/10.34706/DE-2021-02-05

[3] Allam Z. On Smart Contracts and Organisational Performance: A Review of Smart Contracts Through The Blockchain Technology. *Review of Economic and Business Studies*. 2018;11(2):137-156. https://doi.org/10.1515/rebs-2018-0078

[4] Zheng Z., Xie S., Dai H.-N., Chen W., Chen X., Weng J., Imran M. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*. 2020;105:475-491. https://doi.org/10.1016/j.future.2019.12.019

[5] Sarwar M.I., Maghrabi L.A., Khan I., Naith Q.H., Nisar K. Blockchain: A Crypto-Intensive Technology – A Comprehensive Review. *IEEE Access*. 2023;11:141926-141955. https://doi.org/10.1109/ACCESS.2023.3342079

[6] Huang Y., Wang R., Chen X., Zheng Z. Ethereum Transaction Replay Platform Based on State-Wise Account Input Data. *IEEE Transactions on Services Computing*. 2024;17(5):2404-2416. https://doi.org/10.1109/TSC.2024.3390433

[7] Zhikharev A.G., Kidanov V.V., Korsunov N.I. System-Object Modeling of Smart Contracts. *Economics. Information technologies.* 2023;50(4):859-872. (In Russ., abstract in Eng.) https://doi.org/10.52575/2687- 0932-2023-50-4-859-872

[8] Ivanov N., Yan Q., Kompalli A. TxT: Real-Time Transaction Encapsulation for Ethereum Smart Contracts. *IEEE Transactions on Information Forensics and Security*. 2023;18:1141-1155. https://doi.org/10.1109/TIFS.2023.3234895

[9] Shi J., Li R., Hou W. A Mechanism to Resolve the Unauthorized Access Vulnerability Caused by Permission Delegation in Blockchain-Based Access Control. *IEEE Access*. 2020;8:156027-156042. https://doi.org/10.1109/ACCESS.2020.3018783

[10] Zhang Z., Zhang B., Xu W., Lin Z. Demystifying Exploitable Bugs in Smart Contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE Press; 2023. p. 615-627. https://doi.org/10.1109/ICSE48619.2023.00061

[11] Chaliasos S., Charalambous M.A., Zhou L., Galanopoulou R., Gervais A., Mitropoulos D., Livshits B. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners? In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). New York, NY, USA: ACM; 2024. Article number: 60. p. 1-13. https://doi.org/10.1145/3597503.3623302

[12] Samreen N.F., Alalfi M.H. Reentrancy Vulnerability Identification in Ethereum Smart Contracts. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). London, ON, Canada: IEEE Press; 2020. p. 22-29. https://doi.org/10.1109/IWBOSE50093.2020.9050260

[13] Liu C., Liu H., Cao Z., Chen Z., Chen B., Roscoe B. ReGuard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18). New York, NY, USA: ACM; 2018. p. 65-68. https://doi.org/10.1145/3183440.3183495

[14] Qian P., Liu Z., He Q., Zimmermann R., Wang X. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access*. 2020;8:19685-19695. https://doi.org/10.1109/ACCESS.2020.2969429

[15] Xue Y., Ma M., Lin Y., Sui Y., Ye J., Peng T. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, VIC, Australia: IEEE Press; 2020. p. 1029-1040.

[16] Kannengießer N., Lins S., Sander C., Winter K., Frey H., Sunyaev A. Challenges and Common Solutions in Smart Contract Development. *IEEE Transactions on Software Engineering*. 2022;48(11):4291-4318. https://doi.org/10.1109/TSE.2021.3116808

[17] Lebed S.V., Namiot D.E., Zubareva E.V., Khenkin P.V., Vorobeva A.A., Svichkar D.A. Large Language Models in Cyberattacks. *Doklady Mathematics*. 2024;110(Suppl 2):S510-S520. https://doi.org/10.1134/S1064562425700012

[18] Xia S., He M., Shao S., Yu T., Zhang Y., Song L. SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models. *arXiv*:2502.07644. 2025. https://doi.org/10.48550/arXiv.2502.07644

[19] Fekih R.B., Lahami M., Bradai S., Jmaiel M. Formal Verification of ERC-Based Smart Contracts: A Systematic Literature Review. *IEEE Access*. 2025;13:11396-11422. https://doi.org/10.1109/ACCESS.2025.3527158

[20] Wang C., Zhang J., Gao J., Xia L., Guan Z., Chen Z. ContractTinker: LLM-Empowered Vulnerability Repair for Real-World Smart Contracts. In: 2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE). Sacramento, CA, USA: IEEE Press; 2024. p. 2350-2353.

[21] Wei Z., Sun J., Zhang Z., Zhang X., Li M., Hou Z. LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection. *arXiv:*2410.09381. 2024. https://doi.org/10.48550/arXiv.2410.09381

[22] Li Z., Li X., Li W., Wang X. SCALM: Detecting Bad Practices in Smart Contracts Through LLMs. *arXiv:*2502.04347. 2025. https://doi.org/10.48550/arXiv.2502.04347

[23] Namiot D.E., Sukhomlin V.A. On Cybersecurity of the Internet of Things Systems. *International Journal of Open Information Technologies.* 2023;11(2):85-97. (In Russ., abstract in Eng.) EDN: DCDCHM

[24] Namiot D.E., Kupriyanovsky V.P. Web3 Architectural Models. *International Journal of Open Information Technologies.* 2024;12(2):84-95. (In Russ., abstract in Eng.) EDN: CDXZIS

[25] Kupriyanovsky V.P., Alenkov V.V., Sokolov I.A., Zazhigalkin A.V., Klimov A.A., Stepanenko A.V., Sinyagov S.A., Namiot D.E. Smart infrastructure, physical and information assets, Smart Cities, BIM, GIS, and IoT. *International Journal of Open Information Technologies*. 2017;5(10):55-86. (In Russ., abstract in Eng.) EDN: ZISODV

Submitted 25.02.2025; approved after reviewing 28.03.2025; accepted for publication 16.04.2025.

About the authors:

Andrey V. Chaheev, Director of the Cybersecurity Department, Sberbank of Russia (19 Vavilova St., Moscow 117312, Russian Federation), **ORCID: https://orcid.org/0009-0003-2299-5416**, AVChaheev@sberbank.ru

Zakhar R. Nazarov, Master degree student of the Cybersecurity, which is a joint Academic Program with Sberbank, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation); Information Security Analyst of Cybersecurity Department, Sberbank of Russia (19 Vavilova St., Moscow 117312, Russian Federation), **ORCID: https://orcid.org/0009-0004-5276-4253**, zaxar12102001zaxar@gmail.com

All authors have read and approved the final manuscript.